1985

High level language code images for read only memory

Peter Eklund
University of Wollongong, peklund@uow.edu.au

HIGH LEVEL LANGUAGE CODE IMAGES FOR READ ONLY MEMORY

Peter EKLUND

Department of Computer Science,
The University of Wollongong.

Submitted in partial fulfillment of the requirements of CSCI401 Computing Science IV (Honours) at the University of Wollongong in 1985.

ACKNOWLEDGMENT

I would like to thank my supervisor Gary Stafford for his support, assistance and patience over the course of the preparation of this work.

HIGH LEVEL LANGUAGE CODE IMAGES FOR READ ONLY MEMORY

Peter EKLUND

Department of Computer Science,
The University of Wollongong.

ABSTRACT

In real time systems controlled by microprocessors, programs are often stored in ROM. The nature of ROM means that certain restrictions are made. For example self-modifying code is not possible and compile time initialisations of data variables are difficult since the data segment of the software is not included in the ROM code image.

These restrictions manifest themselves in the developed software so that the programmer must at all times be conscious of the memory configuration of the destination machine. This restricts the natural and intended usage of many High Level Languages.

This paper highlights these and other problems associated with code images suitable for ROM and offers a solution to these problems which is both more efficient than the current method of dealing with executable code images in RAM as well as ROM and is easily portable to any UNIX* development environment and associated destination architecture.

*UNIX is a Trademark of Bell Laboratories.

TABLE OF CONTENTS

1. INTRODUCTION

Software development in recent times has increasingly become more micro-computer oriented. That is to say that more and more demands are being made for business, office and engineering tools which run on personal machines.

The limited nature of computer resources in terms of memory and computing power on personal machines has meant that a greater utilization of these resources is called for and this has led to the demise of the software development strategy where by software for the machine is developed on the machine. On-machine software development has taken a back seat to cross-machine development.

Software development for micros is no longer desirable on the micro itself. Development tools use too many of the resources that can be better used in the final product, and such tools themselves are not needed in the finished product.

Bearing this in mind, one question is certain to be raised. How are Software Houses developing applications for the Personal Computer market ?

Super micro facsimiles of the destination machine are one answer for which a good example is the Apple Macintosh XL*.

Another solution which is more practical to the modern per-

*Macintosh is a Trademark of the Apple Computer Corporation

ceptions of Software Development, and particularly the real world of divide and conquer computing, is that of using a time-share environment like UNIX* or VMS# to develop code images which are executable on the destination machine.

This allows the developer to fully utilize the computing resources of the destination machine at run-time while at the same time having the ability to use shared code libraries, storage facilities, and development tools like editors and compilers of the larger host machine.

This involves the use of cross-compilers/assemblers to produce executable code images which can then be loaded into the destination machines memory (i.e. by serial line or disk).

Much the same approach is applied when developing code images for ROM (Read Only Memory). Programs are written on the host machine, cross-compiled and then loaded into the destination machine. This process can be done in one of two ways, either executable code images can be loaded into the RAM of the destination machine from a disk or serial line or alternatively the code image can be burnt into an EPROM or similar erasable non-volatile memory. Once such programs are fully tested on the destination machine they are burnt into a non-erasable ROM which can then be mass produced.

The process by which ROM executable code images are

*UNIX is a Trademark of Bell Laboratories.
#VMS is a Trademark of the Digital Equipment Corporation

developed seems simple enough but in reality there are a number of important considerations which must be fully appreciated by the programmer before success can be guaranteed.

This document formalises these considerations and offers a solution with a degree of transparency to the differences between a conventional executable code image running in volatile memory and that of the executable code that runs in non-volatile memory.

2. BACKGROUND

The conceptual separation of the instructions and data in a program have an intellectual elegance in the explanation of the way in which programs can be understood and written.

Such separation is also as a consequence of the historical evolution of computer machinery. Early machines used external devices, like paper tape, to store the instructions of a program, with internal memory being reserved for program variables. As a result the distinction between instructions and data variables was enforced by hardware constraints.

The introduction of the Von Neumann architecture made such discrimination less distinct. Instructions and program data variables were now both stored internally in volatile memory. However, the conceptual difference between instructions and data was certainly still useful. For example, keeping account of the different address spaces for instructions and data provide a means by which an operating system can detect attempts to write over a programs instructions and thus ensure the integrity of running programs.

As a result, the separation of instructions and data variables addressed to separate spaces within memory, is the usual method of dealing with executable code images.

This observation allows the abstraction back to hardware constrained memories (ROMS).

Such an abstraction needs to be clearly defined. The traditional definition of program instructions as non-volatile and program variables as volatile needs to be elaborated and redefined for the purposes of accommodating modern High Level Programming Languages and Programming Techniques.

3. ROM APPLICATIONS

3.1. SOME EXAMPLES

There are literally thousands of examples of working ROM code. Consumer electronic products like hi-fi, microwave ovens and motor vehicles are some examples where ROMS are used. The complexity of applications varies significantly, controlling room frequency equalisations using signal processing techniques, trivial timing operations based on weight, and trip computer operations based on input from a variety of analogue and digital devices are a few.

More demanding applications include such things as flight control systems on aircraft, weapons and guidance systems on nuclear warheads, data acquisition and production control in industry.

In the personal computer market specialised software like basic interpreters, monitors and operating systems have long been supplied on ROM in various configurations as an alternative to secondary storage.

3.2. STRATEGIES FOR DEVELOPING A ROM CODE IMAGE

There are several strategies for the development of a ROM code image in a HLL (High Level Language) like "C".

1. Discard the program data segment entirely and do not use any global variables. If storage is required at run time use the stack. Pass lots of parameters. Remember data space for global variables is unavailable.

2. Reserve an area of RAM for working storage of the program. The initial state of such an area will be undefined so it is necessary to initialise program variables in code at run time. No compile time initialisation is possible.

3. The ROM image contains both a copy of the data segment at compile time as well as an image of the executable code of the application. When the program is run the data image in ROM is copied to RAM.

All these methods involve a degree of run-time overhead.

In the first case, parameter lists to functions are generally large in non-trivial programs. The overhead necessary to call functions and pass these parameters would no doubt effect system performance. Excessive passing of variables which are unused in functions at that level, but passed and used at a lower level, leads inevitably to programming errors.

The second point, where run-time initialisation of data variables is necessary, the overhead is not drastic and there are certainly no long term performance problems but it does present a degree of sluggishness when the system is first initialised to run. Having to initialise program data variables at

the right place, at the right time, may affect the modularity or structure of the program since the scope of variables may need to be increased to accommodate such processing.

The problems with the third alternative of copying the data segment to RAM from ROM at start up time, are simple enough. Such a scheme wastes ROM space and of course implies the need for a routine that copies the data segment from ROM to RAM.

If a memory configuration has no RAM space (other than that reserved for the stack) then the first option (1.), of using the run-time stack for all program data variables, is the only way to write a program for ROM in a High Level Language.

In almost all cases some working storage RAM can be spared for the program (perhaps at the expense of the run-time stack) and the second option (2.), of run-time initialisation of program data variables, is possible.

In this paper the assumption that is made with reguard to the available RAM memory is that a portion of RAM will be available as working storage data space to the program.

As a consequence of the above, the definition of what a ROM executable code image is, is somewhat like that of option three (3.), except that the ROM image of the data segment will be largely reduced in size by the elimination of all but non-zero values and the code to copy the ROM image of the compile time data segment from ROM to RAM will be supplied automatically. Additionally the programmer will have the option within the High

Level Language to consciously decide what should belong in ROM in terms of both code and data constants as well as what should belong in RAM in terms of dynamic variables and alterable code.

4. HOW A LOADER WORKS

4.1. BACKGROUND

Before a detailed discussion of the operation of the rld (the rom loader) it is necessary to review the operation of a convention loader (like ld).

4.1.1. THE OBJECT FILE FORMAT

The object files (or .o's) that are input to the loader (ld) process have the following format which is consistent with implementations of UNIX on PDP-11, VAX and Perkin Elmer machines.

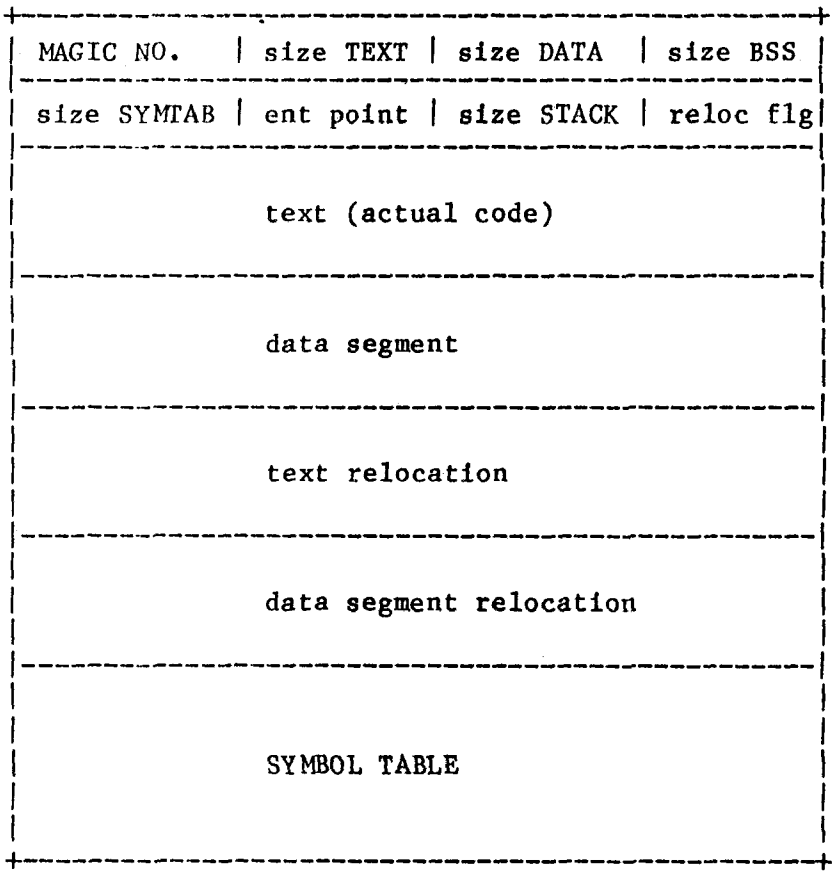


Figure 4.1 - Object File Format

4.1.1.1. MAGIC NUMBER

This is essentially a means by which a file in object format can be determined to be different from printable text files and is a means of type identifying a file.

4.1.1.2. SIZES OF SEGMENTS

The sizes of the text, data, basic stack segments and the symbol table for this .o are compiler generated.

4.1.1.3. ENTRY POINT

In "C" this refers to the address within this .o's text segment that the symbol "_main" is located, if there is no "_main" symbol in this .o then the entry point is zero.

4.1.1.4. SIZE OF STACK

The run time stack size is system dependent.

4.1.1.5. RELOCATION FLAG

If relocation information still exits within this .o then this flag is set (i.e. for a.out this flag will be zero).

4.2. LOADER (ld) PASS ONE

Each of the .o files or archive files have their symbol tables scanned.

The symbols are hashed into a symbol table which is internal to the program. If a new symbol is encountered (one for which no table value exists) then the symbol is added to the internal symbol table.

Once all .o files symbol tables have been scanned the type and value of each of the symbols in the internal symbol table should be known.

If at the end of the first pass of the .o files the internal symbol table contains entries which are not properly defined then at this stage the loader reports the familiar error;

"Symbol xxxxx undefined"

4.3. BETWEEN LOADER PASS 1 and PASS 2

Having examined in the first pass all the .o files the size of each of the .o's segments (text, data and basic stack) were accumulated.

All the symbols in the internal symbol table are now relocated with reference to the origin of the particular segment to which they refer.

```
text_origin = entry point of program
data_origin = text_origin + Text_size
bss_origin =  data_origin + Data_size
```

For example, all symbols in the internal symbol table which

are marked as referring to addresses in the data segment are relocated with reference to the origin of the data segment in the final a.out.

4.4. LOADER (ld) PASS 2

Once again in this pass each of the .o files have their symbol tables scanned.

Each symbol's characteristics are matched against the internal symbol table and if the internal symbol is different, then the symbol in question must be doubly defined. This is reported and the loader stops.

If all symbols check out in a particular .o file they are copied to a symbol table file (which ultimately gets appended to the a.out file if required) and relocation of the program segments can take place.

4.5. RELOCATION

Each word in the text segment is examined simultaneously with the relocation bits which correspond to that word.

If the relocation word is zero then no relocation takes place and the word is appended into a temporary file which will

ultimately represent the text segment in the final a.out file.

If the relocation word is non-zero then it could indicate one of the following;

- a reference to an external symbol (the internal symbol table is scanned and the address of the symbol substituted).

- a reference to an address in the text segment in this .o (i.e. jmp LABEL). The current value of the address following the jmp might be say 10, meaning that LABEL is located at an offset of 10 bytes into this .o's text segment. The value substituted will be 10 + origin of this .o files text segment with respect to the load address of the program.

- a reference to an address in the data segment in this .o (i.e. ldd DATA). The current value of the address following the ldd might be say 30, meaning that DATA is located at an offset of 30 bytes into this .o's data segment. The value substituted for DATA will be 30 + origin of this .o files data segment with respect to the load address of the program. The following assembler listing illustrates;

Motorola 6809 Assembler Sample 6809 Relocatable Code

```
 9   000E                        start:
10   000E    BF   0006R                 stx     Emax
11   0011    301F                       leax    -1,x
12   0013    BF   0002R                 stx     Emin
13   0014    6E   0000R                 jmp     start
23   0028                               .data
26   0002         0000           Emin    fdb     0
26   0004         0000           _Emin   fdb     0
28   0006         0000           Emax    fdb     0
28   0008         0000           _Emax   fdb     0
```

Figure 2 - Sample Relocatable 6809 Assembler Code

If the start address of the .data segment for this code fragment with respect to the load address of the program is at Hex 800, while the start address of the .text segment is at Hex 40 then the application of the loader would produce a relocated listing of the same code fragment as follows;

Motorola 6809 Assembler Sample 6809 Relocated Code

```
 9   004E                        start:
10   004E    BF   0806                  stx     Emax
11   0051    301F                       leax    -1,x
12   0053    BF   0802                  stx     Emin
13   0054    6E   004e                  jmp     start
23   0068                               .data
26   0802         0000           Emin    fdb     0
26   0804         0000           _Emin   fdb     0
28   0806         0000           Emax    fdb     0
28   0808         0000           _Emax   fdb     0
```

Figure 3 - Sample Relocated 6809 Assembler Code

This process is repeated for all the .o files and ultimately the result is a number of temporary files (one for the

text and data segments, and the symbol table). These are combined and renamed to be the a.out file.

5. THE ROM LOADER (rld)

The following is a discussion of the extra processing performed by the ROM loader over an above those of a conventional loader.

5.1. COMPRESSION OF DATA SEGMENT (1)

To allow compile time initialisations of program variables in a HLL, it is necessary for the ROM image to contain values which correspond to the initialised data.

This is done by including a condensed or abbreviated version of the data segment in the ROM image.

At load time a picture of the relocated data segment is generated into a temporary file which is normally (re ld) included in the a.out image and down loaded into the RAM space of the destination machine.

It should be noted that in any non-trivial software, a great number of the program data variables are uninitialised at compile time and thus have no particular value associated with them.

In this case the loader conveniently gives such variables a value of zero (0) which also corresponds to the most frequently used initial value in most High Level Programming Languages.

---

(1) See Appendix E - Data Segment Compression Code

As a result it can be observed that there are large contiguous areas in the compile time data segment which have no particular value and are set to zero (0) by the loader during the course of producing the executable a.out file.

This leds to the idea that the size of the compile time data segment may be reduced while maintaining the same information content. This can be achieved by the elimination of all but non-zero initial values.

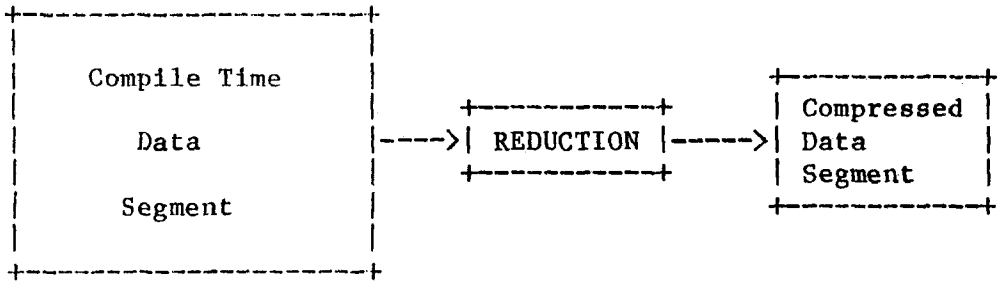


Figure 4 - Data Segment Reduction

As a result the temporary file which represents the compile time data segment is scanned and Data Segment records are set up as follows;

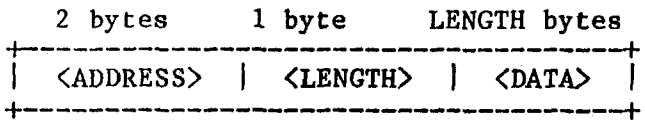


Figure 5 - Compressed Data Records

The data segment is scanned to produce the abbreviated data segment. Since there is a three byte overhead for the setting

up of a new record, four or more bytes of contiguous zeros must be encountered before a new record is needed. If more than 255 bytes of data which contain no blocks of four or more bytes of zeros are encountered then a new record is required.

These records are then appended to the .text segment of the final code image and are included in the ROM image.

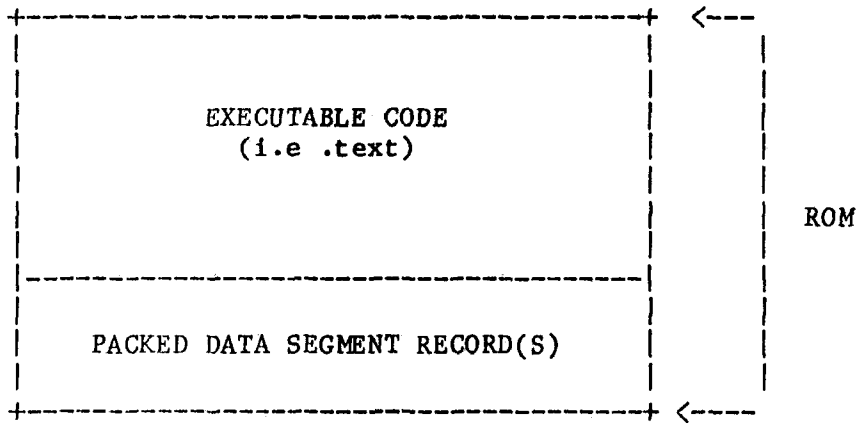


Figure 6 - ROM Image 1

This strategy implies that some code to unpack these records has to be included as the first thing done by the program. Such code will unfold the abbreviated or packed data segment from ROM to the destination machines RAM space.

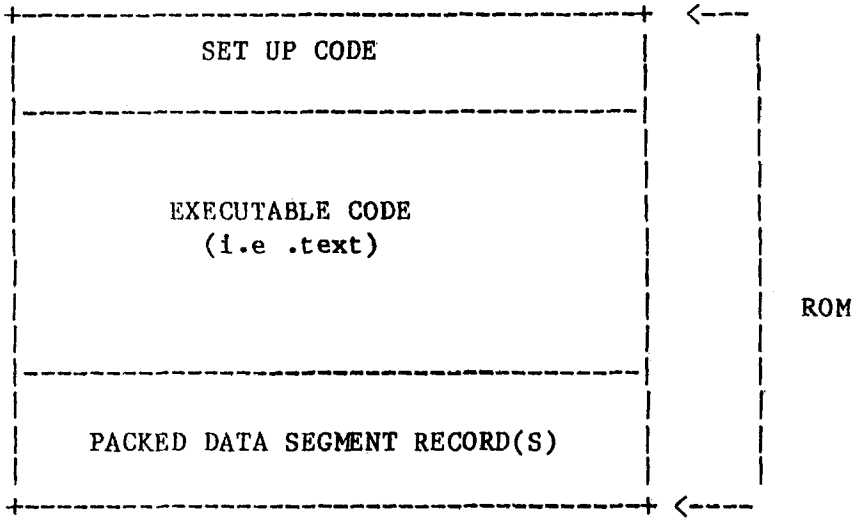


Figure 7 - ROM Image 2

Such set up code is dependent upon the destination machine for which the code is to run it is indicated in a "configuration" file which serves to describe the extents of ROM and RAM in the address space of the destination machine as well as the machine for which the code is being written.

The Setup code (2) indicated must perform the following;

1. Zero out all RAM address space onto which the data segment of the program is to be mapped.

2. Unpack each of the compressed data records into the address space reserved for the programs data segment.

3. Stop once a packed data record is encountered which has a length of zero.

(2) see Appendix A - Setup Code For Other Machines

4. Jump to the programs entry point (i.e. _main).

5.2. ESTIMATION OF COMPRESSED DATA SEGMENT SIZE (3)

There are a number of considerations with regards to where ROM and RAM begin and end in the address space of the destination machine.

Ultimately the start of the code segment will be an address in the ROM space of the destination machine and likewise the address which is the start of the programs data segment will be in the RAM space. Great care should be made when specifying the extents of ROM or RAM space for the destination machine.

In a generalised version of the rld a configuration file to assist in the description of the destination machine's memory configuration saves passing too many flags to the rld.

```
[{ ROM <address1> - <address2> }]
[{ RAM <address1> - <address2> }]
        <machine>
```

<machine>.o is the code to zero the data space and unpack the packed data segment records into that space and is clearly specific to the architecture of the destination machine.

The extents of the ROM and RAM space are essential to the rld. They allow it to successfully relocate symbols, data and instructions. It also provides a good constraint check on the size of the executable machine code and data space the program will require. For example if during the course of program

(3) see Appendix F - Compressed Data Segment Estimation Code

development the code segment of the application becomes too large for the ROM space(s) then the rld can let you know. Likewise for the size of the data segment.

When testing a program by running it from RAM it is obviously unnecessary to specify the ROM space. It is not possible to Down Line Load into ROM. The idea is that program testing and debugging occurs in the RAM space of the destination machine and when the correctness of the code has been demonstrated the relocatable object fragments that make up the executable code image can be run through rld with the extents of ROM being given.

This presents two possibilities for the "configuration" file during program development.

1. No information is given as to the extents of ROM or RAM in the address space.

2. Only the RAM address space is supplied.

In the first case the program will be addressed to location zero in the address space (i.e. RAM starts at location zero) or for the later, the start of the program will be addressed to the beginning of the RAM extents given (note that if both code and data space are too large for the RAM address space in the destination machine testing a program with this memory configuration in RAM becomes an impossible task).

The situation where by the rld is not given separate address spaces for code and data segments presents a large dif-

ficulty so far as relocation is concerned.

There are many possibilities with regard to the arrangement of the program in the destination machine's memory.

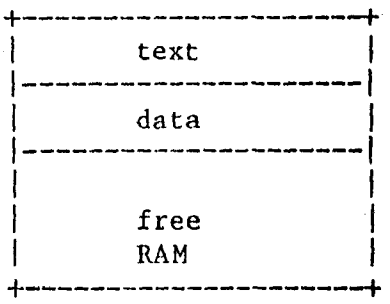


fig. 8.1

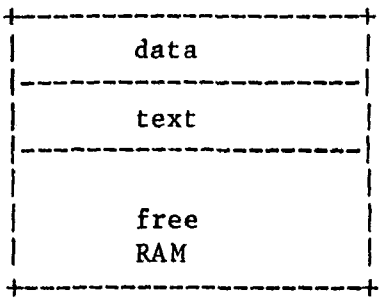


fig. 8.2

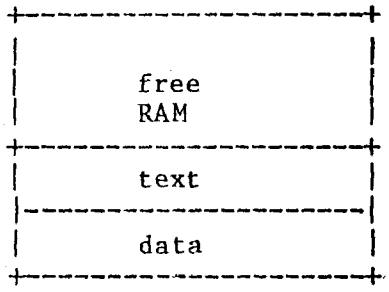


fig. 8.3

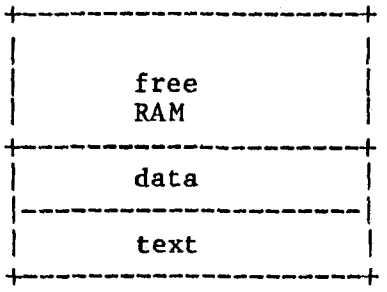


fig. 8.4

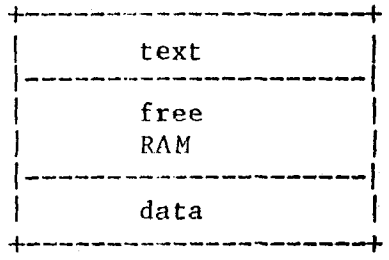


fig. 8.5

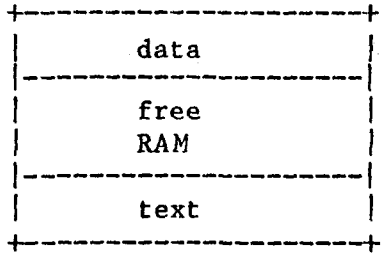


fig. 8.6

Being unconcerned with specifying the start address of the data segment, any of the above arrangements can be chosen as a satisfactory solution depending upon the memory configuration of the destination machine. In some cases one solution will be more suited than others, at other times many possibilities may

present themselves on a given machine architecture.

The most difficult of these possibilities is where the data segment immediately follows the code segment in memory. The reason being that the code segment size is not just the accumulated size of each .o's .text segment but also includes the size of the packed data segment records which will ultimately be appended to it. This presents problems;

1. The start of the data segment in the address spaces cannot be determined until the packed data segment records are generated.

2. The packed data segment records can't be generated until the data segment image is generated by the rld in a temporary file.

3. The data segment image in the temporary file cannot be generated until the origin of the data segment is known.

The argument is circular and the solution is another pass of each of the input .o files so that the size of the packed data can be estimated. Once an estimate is arrived at then the data segment's origin can be defined. The data segment image generated into the temporary file and then the packed data segment records determined.

.data origin = .text origin + .text size + Estimate Pkt_Ds_size

The way in which such an estimate is arrived at needs to be elaborated. As seen in figure 1. (page 10) each object file in

relocatable format (.o) has a relocation segment for each of the data and text segments. Each of these relocation segments is of the exact same size as the segment to which it refers. The relocation segment which corresponds to the data segment will be the same size as the data segment in any one particular .o file and likewise for the text segment.

We are dealing exclusively with the estimation of the compressed data segment and as a consequence will confine the discussion to the relation between the data segment and it's relocation segment although the following equally applies to the text segment.

Each word in the data segment of the object file relocation segment corresponds to a word in the data segment at similar offset. Consider the relocation word to be an attribute or a functional operator on the data segment word. Then if X is the word in the data segment then there exist a Xr such that Xr applied to X yields an Xa which is the corresponding word in the data segment of the final code image or a.out file.

The following is a list of possibilities for the meaning of the Xr operator:

1. X is a reference to an external symbol in which case Xr indicates that the internal symbol table of the loader would be searched for the final Xa.

2. X is a reference to an local symbol in which case the internal local symbol table of the loader would be searched

for the final Xa value.

3. A reference to a label in the data segment within this particular code fragment in which case the loader discovers a value for Xa once X is added to the origin of the data segment of this code fragment in the final a.out file.

4. X is a reference to a label in the text segment within this particular code fragment, the loader calculates a value for Xa by adding X to the origin of the text segment of this code fragment in the final a.out file.

5. X is a reference to a label in the bss segment within this particular code fragment in which case the loader discovers a value for Xa once X is added to the origin of the bss segment of this code fragment in the final a.out file.

6. No relocation is necessary then Xr has a value of zero (0) and no action is taken by the loader. The Xa is the same as X.

The data segment reduction process eliminates zero (0) values of Xa. To produce a compressed or abbreviated data segment the estimation of its size will be is based on a prediction for Xa without knowledge of absolute addresses.

If Xa is predicted to be zero then such an Xa will not appear in the compressed data segment in the final a.out file. It can therefore be discounted from the estimation of the size of the compressed data segment.

Such an estimate of the compressed data segment size is described by the following;

1. for each of the .o files input to rld

    1.1 examine each word in the .o data segment

    1.2. examine the corresponding word in the .o data relocation segment

    1.3. if both are zero then the corresponding word in the a.out data segment will be zero so discount this word.

This process yields an over-estimate of the compacted data segment since the application of some relocation functions Xr to particular X values may produce results of zero (0) for Xa which were not predicted.

5.3. A NON-CONTIGUOUS ADDRESS SPACE

Contiguous address spaces of ROM and RAM in the destination machine are ideal, unfortunately in the real world they are not always the case.

On chip RAM or ROM is often not contiguous in the address space with external ROM and RAM chips. Some micro manufacturers have set a precedent of configuring screen memory in the middle of RAM space.

Some machines have an address space which may be switched between either ROM or RAM. (4)

This means that it is necessary to consider the problem of building a code image into these non-contiguous areas.

The problem introduces a number of complexities to the rld.

1. Splitting object code images - difficult (although possible) to start splitting any one particular code image especially within a single instruction boundary.

2. Relocation difficulties - relocating particular code fragments once a decision has been made as to were various parts belong.

3. Juggling code such that it fits into the memory configuration in the best way possible.

---

(4) Apple IIe Technical Reference Manual Bank Switched Memory pp 68.

The solution to point one (1.) is easy, under certain assumptions.

For simplicity's sake and because of the ease by which the programmer can split a source file into several smaller source files and then re-compile, it is expedient that the boundaries by which the programs executable code be split, if necessary, on a .o by .o basis.

This circumvents the problem of splitting single instruction and allows the rld to invoke a warning if the .o code image becomes two large for any one contiguous ROM space and a similar message if the data segment of any particular .o is in itself too large for any one RAM space.

Consider the following memory configuration;

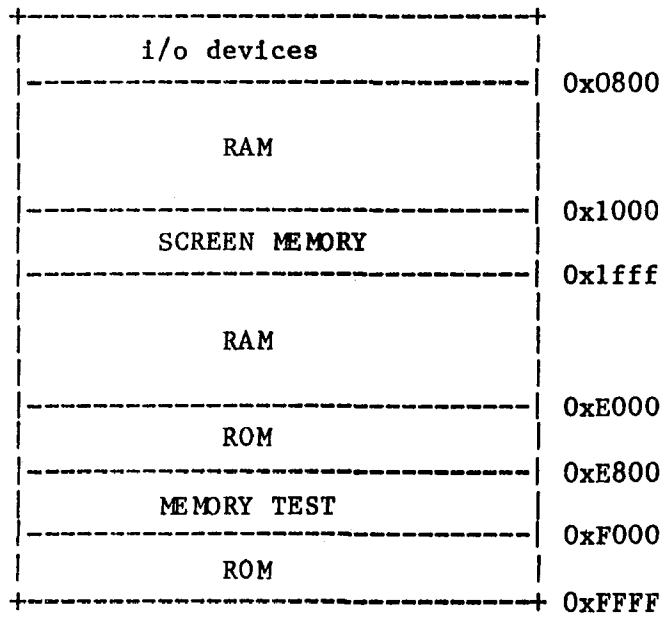


Figure 9 - Sample Memory Configuration

In the case of this memory configuration a "configuration" file could look like the following for a program under development for the M6809 using a generalised rld;

```
ROM e000 e7ff
ROM f000 ffff
RAM 0800 0fff
RAM 2000 e000
6809
```

Figure 10 - Configuration File for Sample

So two extents of ROM and RAM are given. If any one .o file has a .text segment which exceeds the largest extent of ROM space that has been specified then the rld will indicate this. Likewise if any one .o file has a .data segment which exceeds the largest contiguous extent of RAM space a similar message would be produced. Checks on the overall size of the machine code image are still maintained and if the overall address space of the program exceeds the limitations of either ROM or RAM then this will be indicated.

At the same time, the rld added "unpack and clear mem code must be flexible enough to allow for widely spread area memory being cleared at program startup.

Of course there are some limitations introduced with reguard to the number of extents of ROM and RAM that are given to the rld, and for the purposes of this exercise, twenty (20) of each will suffice.

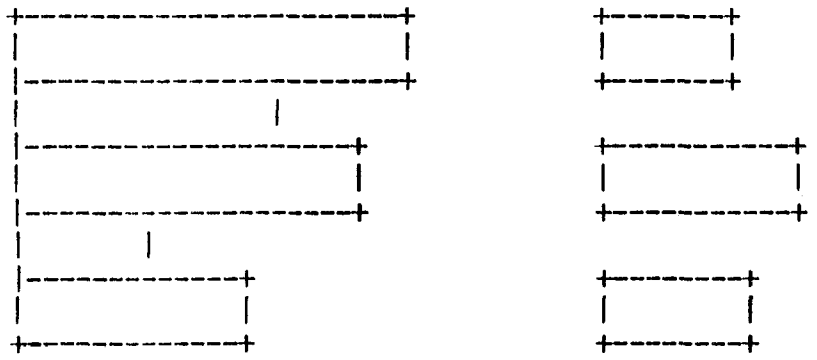
5.4. THE KNAPSACK PROBLEM

The allocation of the fit of parts of the overall code image into the address spaces provided is a somewhat simplified version of the Knapsack problem.

The Knapsack problem is to optimally fit a number of items into a Knapsack. The possibility exists that only one combination of items in the sack results in a solution (A sack full of all items).

In this application, the objects that are to be fitted (Pieces of relocatable code) have a single attribute or dimension of "size". In reality, the Knapsack problem must take into consideration the problem of dealing with fitting objects based on the dimensions of length, breadth and width. Some objects may contain other properties, for example a tea pot is clearly a three dimensional object which not only consists of width, breadth and a length but has a further property relating to it's unique shape. If the tea pot contains water then it has yet another property which clearly affects the position in the Knapsack which it can occupy, so that no water is spilt.

The application of the Knapsack analogy yields a Knapsack which in this case is actually the fragmented address space of the destination machine. The items to go into the Knapsack are the relocatable executable code fragments split on .o file boundaries.



KNAPSACK
(fragmented address space)

ITEMS
(executable machine code image fragmented on .o file boundaries)

Figure 11 - Knapsack

This problem has worst case behavior of n! (n factorial) complexity. This means that if there are n items that a worst case behavior of an algorithm to solve the problem will be to generate n! (n factorial) permutations of the items. A problem of such computational complexity is considered indeterminate.

However, given that the number of items that are to arranged is small, (usually no more than twenty .o files could be imagined) and that the number of contiguous memory areas that will be specified will also be small, (five or six extents of address space would be highly unlikely ,in this case restricted to twenty (20)), then the worst case behavior would be 20! or 2.43 E 18 tries to find a successful combination of .o items or determine that the problem was unsolvable, is not unreasonable.

As a consequence of the relatively small size of the numbers of object code fragments. The likelihood of exactly one

fit of object code fragments into a machines memory configuration is remote, so the worst case behavior is seldom approached.

5.4.1. FITTING CONDITIONS

There are a number of checks that can be applied to the machine code fragments and the memory spaces into which they must fit. The rld must apply these checks to insure that a fit is either possible, and then relocate object code fragments accordingly, or alternatively indicate which non-fit condition holds.

The simpliest checks are for excessive size.

5.4.1.1. SIZE COMPATIBILITY

Clearly if the sum of the code fragments sizes exceeds the total available memory space then there is no possibility of a fit condition. The rld indicates that no fit condition holds for this machine code image in this memory configuration.

5.4.1.2. FRAGMENT SUITABILITY

Having satisfied the size compatibility criteria the next check is just as simply and intuitive. If any one particular code fragment exceeds the size of the largest memory space

available then a fit is not possible and further splitting of source files to yield smaller relocatable object fragments is recommended.

5.4.1.3. OTHER NON-FIT CONDITIONS

A good example of a of a non-fit condition which is difficult to detect is when two machine code fragments have a size which is greater than all but a single memory space and that memory space is too small to accommodate both fragments. Consider the following;

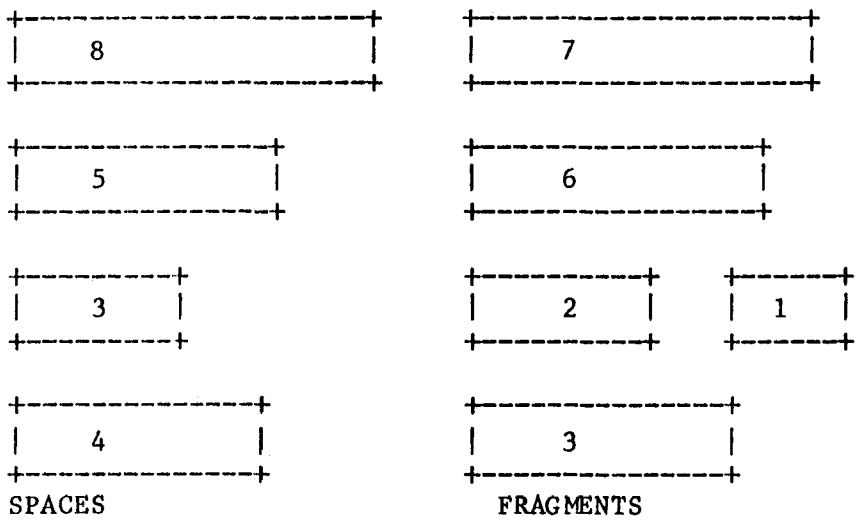


Figure 12 - Sample Knapsack Problem

The simple checks for total size compatibility and fragment suitability prove positive although a fit of the given fragments into the memory spaces is not possible.

The probability of such a non-fit condition based on the

process of normal software development seems low. As a consequence of the general case of a relatively small number of fragments and memory spaces used, the generation of all permutations of ordered fragments into the memory spaces is an acceptable method of determining such a non-fit condition.

If the application of all permutations of code fragments does not yield any solution then the rld can recommend further reducing the size of individual machine code fragments by the process of splitting source code files.

The problem of fitting the blocks of machine code and data into a non-contiguous memory address space can be solved in a reasonable time given the small numbers with which we are dealing, and of course bearing in mind that the problem is solved before starting to move blocks of machine code and data in the machines memory.

5.4.2. FINDING A SOLUTION (5)

Of special consequence to the rld is the order in which the relocatable object files have been passed to it. The assumption made is that the rld user may be well aware of both the size and the memory configuration being used. As a consequence the object files have been passed to the rld with the knowledge that the

(5) See Appendix G - Knapsack Solution

order and position of those fragments given is the desired outcome for a fit into the destination machines memory spaces.

As a result the rld's first attempt at a solution to the Knapsack problem is an attempt to "fit" the relocatable object code fragments in the memory space configuration in the order in which they were supplied.

If this fails then a different strategy is developed. As a general rule and although exceptions can be contrived it is possible to say that "large code fragments go into large memory spaces". As a consequence of such an assumption it is possible to approach a solution more rapidly by sorting both the machine code fragments and the address spaces in order of size.

Finally, if sorting fails then the next permutation in the sequence for code fragments will be generated until either a solution has been found or all permutations of the code fragment sequence have been explored.

5.5. SUMMARY OF RLD (6)

In summary then the main jobs of the rld can be described as follows;

1. Collect all symbols for each .o file and report on any which are undefined.

2. If separate Instruction and Data (I/D) spaces have not been supplied then make an estimate of the size of the packed data segment.

3. If more than one ROM or RAM memory space has been supplied then solve the Knapsack problem.

4. Relocate symbols.

5. Relocate the text segment with respect to the supplied start of the address space and copy the result into a temporary file.

6. Relocate the data segment with respect to the estimated start of the data address space and copy the result into a temporary file.

7. Compress the data segment temporary file into a packed data segment temporary file.

8. Create the a.out file by concatenating the text temporary file and the compressed data segment temporary file.

---

(6) see Appendix D for manual entry for rld

5.6. PROBLEMS WITH RLD

Three main problems exist which can make the application of the rld less useful. These problems can be easily spotted.

1. If the data segment of the code image (as generated in a temporary file at load time) has no contiguous blocks of four or more zeros then the rld will make no substantial saving to the size of the code image produced. The romable code image may in fact be larger than the a.out produced using your conventional loader. This is extremely unlikely to happen for programs of anything other than a trivial size. Such programs, having been recognised, would be best implemented as ROM images if both the text and the data where on ROM and the data copied to RAM when the program starts. However, the rld provides this feature automatically and thus prevents the need for extra code.

2. If the memory configuration of the destination machine onto which the program is to run has no RAM address space then the application of the rld will not help. The program must be written without global variables and the run-time stack used for all internal program storage.

3. The Implementation is dependent upon the a.out format. This is more a portability consideration than a rebutter of the rld's concepts.

6. ASSEMBLER DIRECTIVES IN THE HLL

The work done by the rld (rom loader) relies on two questionable assumptions concerning the nature of the make up of a program.

1. That all data variables are dynamic and may change or do change over the course or the program's execution and in so doing belong exclusively in RAM.

2. That the .text segment or the executable code of the program is static over execution of the program (i.e. code is not self-modified).

6.1. ASSUMPTION 1 - DYNAMIC DATA

Some variables like error strings, state transition tables, printf strings etc are constant and global throughout the program's execution.

They are however, by convention, included in the .data segment of the a.out image. This leds to a certain amount of wastage and redundancy in the rld produced machine image. Consider the following string;

```
char str[] = "program constant: this is a constant string";
```

Redundancy is introduced here in that this string is actually stored in the machine code image twice at execution time.

The first instance is in the packed data segment records in the ROM image. The second instance is in the RAM space of the program after the packed data records have been unfolded into it.

The result is that we have two occurrences of a constant string which could have just as well been stored in the one place in ROM.

This implies a desire to be able to specify at compile time in the High Level Language what constituents a "constant". There is no such feature for "C" within the language grammar itself that enables such a concept to be easily implemented.

6.2. EXCEPTION RULE ONE - READONLY DIRECTIVE

The readonly directive provides the facility to include in the .text segment variable definitions which previously were prefixed by the assembler pseudo op-code .data.

```
readonly char str[] = "Initialised constant string";
```

This constant string is treated as if it were a code fragment, included and addressed to the .text segment from which it is included into the ROM code image by the rld.

6.3. ASSUMPTION 2 - STATIC CODE.

On the whole this would be a fair assumption. Most High Level programming languages have no facility to alter code at run time simply because it is impossible to take the address of a function or procedure. Because "C" has the facility to take the address of the function and then use type casts to process that address as if it were a variable array, self modifiable code is a possibility although admittedly a rather obscure one.

It becomes very difficult to contrive an appropriate or significant example of a self-modifying code fragment in "C". Here is an example of trivial proportions which serves as a demonstration that self-alterable code is possible.

```
Sleep()
{
        int i;
        printf("Sleeping for %d clock ticks",70);
        for (i=0; i < 70; i++)
                printf(".");
}
In_Sleep()
{
        int i,j,k;
        char *buf;

        buf = &Sleep;
        j = &In_Sleep;
        j -= buf;
        for (i=0; i < j; i++)
                if (buf[i] == 70)
                        buf[i] =255;
}
main()
{
        Sleep();
        In_Sleep();
        Sleep();
}
```

Figure 13 - Self-Modifying Code

Figure 13 - the Sleep() loop control variable (70) is changed by the application of In_Sleep(). In_Sleep() takes the address of Sleep() and type casts to process the function as if it were an array of characters. Whenever the value 70 is encountered it is replaced by 255. Such processing is dangerous especially if 70 is a valid machine op-code used in Sleep().

As a consequence of the possibility of self-altering code, it is necessary to be able to specify at compile time whether or not a function(s) belong in RAM.

6.4. EXCEPTION RULE TWO - "RAMON" AND "RAMOFF" DIRECTIVES

All functions and code are presumed to be suitable for ROM unless the "ramon" directive is encountered, at which point all code and data which follows is presumed to be located in RAM until a "ramoff" directive is encountered.

```
ramon Sleep()
{
        int i;
        printf("Sleeping for %d clock ticks",70);
        for (i=0; i < 70; i++)
                printf(".");
}
ramoff
```

Figure 14 - Using Ramon and Ramoff

6.5. COMMENT

Generally, the assumption that self-modifying code is not allowed, is acceptable to most High Level Languages. This being the case all executable code can automatically be located in ROM.

The first assumption is more readily challenged since there is a genuine need to specify program constants so they may be included in the program's ROM image.

The introduction of the compiler directives which translate to assembler directives involved only minor changes to the "C" compiler (6) and no changes to the "C" grammar.

---

(7) see Appendix D - Changes to "C" compiler.

7. RESULTS

The comparison of code images produced by the RAM loader (rld) and the conventional loader (ld) are difficult to measure both in terms of size and performance.

7.1. PERFORMANCE

A logical state analysiser is needed to correctly measure the differences in speed and performance between executable code images produced by the rld and those that produce ROM executable code images produced using more conventional means as described in Section 3.2 (page 6).

7.2. EXAMPLE - THE TERMUX

The TERMUX code is a Message Passed Operating System for a terminal multiplexier. Its function is to control the access of several UNIX terminals to a single node on a Cambridge Ring Local Area Network. This is a good example of a piece of code ideally suited to be placed in ROM. This would prevent the need to Down Load the code each time the Cambridge Ring Network was restarted.

The code was not suited for ROM unless considered in its

entirety (i.e. .text + .data segments with the data segment copied into RAM at run-time) but was restartable after reset i.e. it re-initialised its own modified data space. The size of this code was 15.2K bytes.

Without change to the source, the programs object files, as produced by M6809 "C" compiler, were run through the rld and as a consequence the machine load image was reduced in size to 13.2K bytes.

After modifying the source code so that compile time initialisation were included and run-time initialisations eliminated the load image was further reduced to 12.3 K bytes, which represents a real saving of 0.9 K, and an improvement over the first figure in the order of % 20 in the size of the machine code image. It is clearly more efficient to initialise data space as a complete entity rather than variable by variable.

A ROM executable version of the TERMUX multiplexier code could be developed using conventional software tools by using one of methods described in section 3.2. Such a code image would be expected to be in the order of 17-20k bytes.

An improvement in the size of executable machine image of a program in the order of 20% implies a real saving in the amount of ROM space that is used.

7.3. CODE IMAGE SIZE

Generally the physical size of the file that contains the compile time code image will be reduced in the order of 20% over executable code images produced using conventional software tools.

The side effects are that in all computing systems where disk space is at a premium, the savings made in using the method outlined to reduce the amount of disk storage consumed by executable code images, may be in itself a sufficient reason to introduce such a scheme. Likewise load time of executable images from secondary storage will be reduced.

7.4. PROGRAMMING STYLE

Qualitative improvements are a little more difficult to judge. Clearly the rld allows for compile time initialisations of internal program variables. Global variables are allowed and generally the programmer is freed from the house keeping that usually goes along with development of ROM executable code in High Level Languages using conventional tools. Programming style is unaffected and existing software can be easily ported for ROM.

7.5. SERIAL RESTARTABLE CODE IMAGES

One other important side effect is that the format of the executable image implies that all programs will be serially restartable. This is of particular significance in debugging. After having changed many of the programs internal variables, re-execution of the machine image will automatically re-initialise program variables by unfolding the packed data segment into the programs address space. This leaves the tester with an apparently fresh executable image as if (in conventional terms) it had been re-loaded off a disk or Down Loaded to the machine. Once again such a convenience saving is hard to quantitatively measure.

8. EXTENSIONS

8.1. IMPLEMENTATION FOR OTHER ARCHITECTURES

The worth of such a system is only really as good as the ease of which it can be ported to suit other machines, and this would be true for both destination and host machines.

The main portability consideration of a generalised rld for a variety of destination machines is the start-up code which zeros the program data space and unpacks the compressed data segment into RAM. (8)

In terms of the portability such a rld to run on a number of machines the main consideration is the format of object or .o files used on that machine. The expected input of a down loader or ROM burner and the byte order of the host machine.

8.2. ASSEMBLER DIRECTIVES IN OTHER HIGH LEVEL LANGUAGES

Consideration should be given to developing another language implementation of the assembler directives that have been described above for "C" (READONLY, RAMON and RAMOFF). Given the ease by which the M6809 "C" compiler was changed to accommodate these directives (9) this is not seen as a difficult procedure.

(8) See Appendix A for Set Up code for Motorola 6809, 68000, Rockwell 6502, Intel 8088 and Bill.
(9) See Appendix C - Changes to the "C" Compiler

9. CONCLUSIONS

High Level Languages are growing in popularity and there is consensus that it is easier to develop software using these languages than assembly language. As a consequence of the ease of development and a general increase in hardware efficiency and memory size, many of the areas which were considered to be the domain of assembly language programs due to the need to contain execution times and machine code size, need to be re-evaluated. Such is the case with ROM applications.

The concepts by which the rld produces executable code images for ROM are easily grasped and can be used to great effect. New assembler directives can be introduced into the High Level Programming Language to facilitate the implementation of these concepts. The resulting changes to the compiler are minimal. Ultimately the savings that can be achieved in the size of executable code images are not insignificant.

Apart from these advantages the benefits of a loader which can produce ROM executable code from a High Level Language, can largely be measured in terms of the reduction in programming effort. The constraints imposed by the memory configuration of the machine for which software is being developed are relieved and no impact upon programming style is experienced.

In this paper many of the concepts that need to be addressed to overcome the problems of coding programs for ROM were discussed and a solution provided which ensures the success

of High Level Languages code images for ROM.

BLIOGRAPHY

OMSI Pascal for ROM Applications J.E.Sanders IEEE Transactions on Nuclear Systems Vol. 29 No. 1 pp 67-70.

Principles of Software Engineering and Design M.V.Zelkowitz A.C.Shaw J.D.Cannon (Prentice-Hall) New Jersey, 1979.

A Language-Oriented Approach to Computer Architecture C.R.Wilcox Technical Report No.191 Stanford Electronics Laboratory, 1980.

6809 Assembly Language Programming L.A.Leventhal (OSBORNE/McGraw-Hill) Berkeley, California, 1982.

6502 Assembly Language Programming L.A.Leventhal (OSBORNE/McGraw-Hill) Berkeley, California, 1979.

6502 Software Design L.J.Scanlon (Howard W.Sam and Co.) Indianapolis, 1980.

Apple IIe Reference Manual Jef Raskin (Apple Computer Inc) Cupertino, California, 1978.

How to Solve it By Computer R.G.Dromey (Prentice-Hall) London, 1982.

Pascal User Manual and Report Kathleen Jensen and Niklaus Wirth (Springer-Verlag), New York, 1974.

AZTEC "C" Programmer Reference Manual Mantex Software Pty Ltd New York, 1980.

The "C" Programming Language B.W.Kernighan, D.M.Richie Prentice-Hall (New Jersey), 1978.

Unix Programmers Manual The University of Wollongong, Seventh Edition, Vol 1, 1982.

GLOSSARY

a.out file - the default name for an executable file generated by a "C" compiler on the UNIX operating system.

archive file - a group of files combine into a single archive file which are generally used as library subroutines as input to a loader.

assembler directives - pseudo operation codes issued to an assembler usually applying to the position of code or data in the programs final address space (e.g. org, obj, .data, .text, .bss).

basic stack segment - uninitialised data is placed into this address space

.bss - assembler directive indicating that the following data or code belong in the basic stack segment and should be addressed with reference to the start of this segment and the definitions which preceded belonging in the bss.

Cambridge Ring - a type of Local Computer Network which makes use of an empty slot Ring Topology.

cross-assembler - a program which takes as input operation codes in an assembler language and generates executable code which may not be usable on the machine on which the assembling process took place.

cross-compiler - a program which takes as input a computer language and generates executable code which may not be usable on the machine on which the compiling process takes place.

cross-machine development - the process of developing computer programs on one machine architecture for execution on some other machine architecture making use of cross compilers and assemblers.

.data - assembler directive indicating that the following data or code belongs in the data segment and should be addressed with reference to the start of this segment and the definitions which preceded belonging in the data segment.

data constants - variables used internally to a program which will never change through any execution path of that program.

data segment - internal program variables which are not placed in the bss segment.

data space - data is placed in this address space, the extents of the addresses that a programs data occupies in the executing program.

destination machine - the machine on which a program which has been cross-developed can be run.

dynamic variables - variables used within the program that change when there exists some execution path within the program that assigns values to that variable.

global variables - variables which are within the scope of any part of the program, i.e. can be changed anywhere throughout the program.

host machine - the machine on which cross-machine development is done.

I/D spaces - Instruction and data spaces, some machines have separate address segments for instruction and data.

initialised data - program variables which are initialised at compile time.

knapsack problem - problem involves the insertion of a number of items into a knapsack or bag which is of an irregular shape such that there may exist at worst only one combination of items into the knapsack which fit.

ld - pneumonic of loader, a UNIX utility which links together several relocatable object or .o files to produce an executable a.out file.

link/editor - IBM terminology of a utility which performs a similar function to that of the UNIX utility ld.

linker - object programs which are produced by an assembler which allows external references can have those references resolved by a linker.

loader - is generally considered to be a program which takes object code from an assembler and places it in memory. In this document loader is a term used to describe a utility which produces absolute code from relocatable object code fragments, resolving external or cross references and "linking" together object code fragments. The term "loader" does not refer to a utility which loads a program into memory.

main (_main) - "_main" is label at which tne entry point of "C" programs is equated to.

modularity - the process of sub-dividing a problem into smaller, more manageable tasks.

on-machine development - the process of writing programs on a machine to be executed on that machine.

packed data segment - a condensed version of the data segment.

RAM - random access memory.

RAM space - the extent of the address space of a machine that is occupied by RAM.

relocate - the process of making executable code run in a different address space.

rld - the name given to the loader which produces code images suitable for execution in ROM.

ROM - Read Only Memory.

ROM space - the address space in a computer memory occupied by ROM.

run-time - the duration of time in which a computer program runs.

self-alterable code - executable code which modifies itself during execution.

source machine - see mother machine.

state transition tables - data in the form of a tabular structure which indicate to a compiler or some other program the state at which the processing has achieved.

super micro facsimiles - computer which is similar to some destination machine in terms of machine code but which is superior in computing resources.

termux - a machine based around a MOTOROLA 6809 processor which is was designed and built at the University of Wollongong to act as a node on a Cambridge Ring Network, typical function which be as a terminal multiplexer.

text - assembler directive indicating that the following data or code belongs in the text segment and should be addressed with reference to the start of this segment and the definitions which preceded belonging in the text segment.

text segment - executable program code in machine language which is static and unchanging over the execution of a computer program.

VMS - a time-sharing operation system developed by Digital Equipment Corporation.

working storage - internal program variables which change over the life time of a programs execution.

APPENDIX A - Setup Code for Other Machines

The following is the startup code to perform the function of zeroing all RAM memory and then unpacking the abbreviated data segment from ROM into the RAM data space.

The code is not difficult to write or modify with respect to the special requirements of the destination machine that is to be used.

Some of the more popular machines have been catered for here. The Intel 8088 (as implemented in the "Port" operating system) , Rockwell 6502 and the Motorola 6809.

```
        title "Unpacking startup code for 6502"
        seg     1
len     ds      1               ; it's length
length  ds      2               ; tdata - bdata
        seg     2
Unpack
startd  equ     Unpack-20       ; start of packed data records
bdata   equ     Unpack-16       ; start extents of RAM
tdata   equ     Unpack-8        ; end extents of RAM
REG     equ     0               ; fudgy pseudo registers for computing
                                ; 16 bit address
where   equ     2
from    equ     4               ; these are necessarily in zero page
```

```
zeroout
        ldy     #0
repeat
        clc
        lda     tdata,y
        sbc     bdata,y
        sta     length
        iny
        lda     tdata,y
        sbc     bdata,y
        sta     length+1        ; length contains the number of
                                ; bytes to be zeroed starting at bdata
        ldx     #0
        cpx     length
        bne     continue
        cpx     length+1
        bne     continue
        jmp     load            ; length is zero so no more zeroing

continue
        ldx     #0
        lda     #0
        lda     bdata
        sta     REG
        lda     bdata+1
        sta     REG+1
zerolp
        sta     ($REG,x)        ; move zeros
        inx
        cpx     length          ; finished ?
        bne     zerolp
        inc     REG+1           ; next 255 byes of address space
        dec     length+1
        lda     length+1        ; any more zeros to move
        beq     load
        ldx     #255
        jmp     zerolp          ; zero out the next 255 bytes
load
        lda     startd          ; where are records ?
        sta     from
        lda     startd+1        ; set them up in from
        sta     from+1
        ldx     #0
again
        lda     ($from,x)       ; first two bytes are address in
        sta     where           ; data segent where this stuff belongs
        inx
        lda     ($from,x)
        sta     where+1
        inx
        lda     ($from,x)
        beq     finish          ; when length =0 finish up
        sta     len             ; the third bye in each rec is the length
```

```
loop
        inx
        lda     ($from,x)       ; move data from to where
        sta     ($where,x)
        cpx     len             ; finished this record ?
        bne     loop
        inx                     ; yes so go set up another record
        jmp     again
finish
        jmp     main_           ; yets go and run the program
        end
```

Bill is a RISC machine which is being designed by Gary Stafford. The following is Bill Assembler to perform the zero and unpack function.

```
_Setup:
        r0 = _Packed_data;          where the packed data segment is
        r1 = _Data_size;            size of the data space
        r2 = 256;
        r1 -= 256;
        r1 >>= 1;

zero_mem:
        *r1 = zero;                 clear word
        r2 += 2; --r1;              incr ptr and dec count
        if r1 != zero
                pc = zero_mem;      zero memory

new_record:
        r2 = *r0;                   get packed count
        if  r2 == zero
                pc = out;           zero implies done
        r1 = r0[2];                 address where it goes
        r0 += 4;                    skip header

another:
        r3 = *r0; r0 += 2;          pickup word and incr ptr
        *r1 =  r3; r1 +=2           putdown word and inc
        --r2;                       dec count
        if r2 != zero
                pc =  another;      keep going

        pc = new_record;            finish with this rec

out:
        pc = _Start_address;        go start program
```

The following code is the implementation of the zero data space and unpack data records on the "PORT" operating system running on Intel 8088 based IMB PC/XT.

```
        origin(  ?[code]);
        pushf;
        cli;
        mov ax,cs;
        mov ds,ax;
        popf;

        mov     cx, $02;        /segment count in the data segment
        shl     cx;             /convert to word count
        shl     cx;
        shl     cx;
        sub     cx, #$08        /skip the first 16 bytes

        move di, #$10           /start of data segment to zero
        sub     ax, ax;

        cld;
        rep     stow;           /zero the data segment

        mov     si, $06;        /start of packed data

get_addr:
        lodw;
        xchg    di;             /load the destination address
        lodw;
        xchg    cx;             /load the byte count
        jcxz    unpack_done;
        rep;
        movb;                   /copy the data
        hop     get_addr;

unpack_done:
        pushf;
        cli;
        mov     ax,ss           /restore original ds
        mov     ds,ax
        popf;

        call    _Call_program;  /call root function
```

The following is the implementation of the zero and unpack code for the Motorola 6809. This code was used in the implementation for an implementation of a Rom Loader (rld) along with cc09 - a "C" compiler for the M6809.

```
        .globl  Unpack
        .text
Unpack:
startd equ      Unpack-4    /* where  are the packed DS's
bdata  equ      Unpack-2    /* where is the start of the DS proper
zeroout:
        ldu     bdata       /* point to where the data space starts
        ldy     #0          /* y is zero
        ldd     bdata
        subd    startd      /* the start of the DS lower in address
        blt     over        /* space than the address of the PDS
        tfr     s,d         /* d = sp
        bra     over2
over:
        ldd     startd
over2:
        subd    bdata
        andb    #x'fc       /* mulitple of four
        tfr     d,x         /* x = d
zerolp:
        sty     ,u          /* move zeros
        sty     2,u
        leau    4,u         /* update pointer
        leax    -4,x        /* dec d
        beq     loadd       /* finish zeroing up to sp
        bra     zerolp
```

```
/*
/*      u is curret position in of packed DS proper
/*      y is address of where data belongs in real DS
/*      d is size of current packed DS record
/*
loadd:
        ldu     startd      /* u points to start of packed DS
again:
        ldx     ,u          /* x address of where data to go
        lda     2,u
        beq     quit        /* size zero so must have finished
        leau    3,u         /* u points to first data to go
loop:
        ldb     ,u          /* load b with 1 byte of data
        stb     ,x          /* store it at location x
        leau    1,u         /* next four bytes of data
        leax    1,x         /* destination address update
        deca
        bne     loop        /* keep moving it
        bra     again       /* size zero so on a new record
quit:
        jmp     __Kernel
        rts
```

NAME

rld09 - link editor for ROMable code

SYNOPSIS

rld09 [-vsulxXrdc] [-o] [name] [-t] [name] file ...

DESCRIPTION

Rld09 combines several object programs into one load module, resolves external references, and searches libraries . In the simplest case several object files are given, and it combines them, producing an object module which can be either executed or become the input for a further rld09 run. (In the latter case, the -r option must be given to preserve the relocation bits.) The output of rld09 is left on a.out This file is made executable if there are no unresolved references, no errors occurred during the load. and the -r flag was not specified.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines satisfying an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries is important.

The symbols "_etext", "_edata", "_bss" and "_end" ("etext", "edata","bss" and "end" in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, the first location above the uninitialized data, and the first location above all data respectively. It is erroneous to define these symbols.

Rld09 understands several flag arguments which are written preceded by a -. Except for -l ,they should appear before the file names.

-s `Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by strip (1). This option is turned off if there are any undefined symbols.

-u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-l This option is an abbreviation for a library name.-l alone stands for /lib/libc.a09, which is the standard system library for C and assembly language programs. -l x stands for /lib/lib x .a09, where x is a string. A library is searched when its name is encountered, so the placement of a -l is significant.

-x Do not preserve local (non- .globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

-X Save local symbols except for those whose names begin with `L' or `F'. This option is used by cc to discard internally generated labels while retaining symbols local to routines.

-r Generate relocation bits in the output file so that it can be the subject of another rld09 run. This flag also prevents final definitions from being given to common symbols, and suppresses the `undefined symbol' diagnostics.

-d Force definition of common storage even if the -r flag is present.

-c Complain about undefined symbols even if the -r flag is present.

-o The name argument after -o is used as the name of the rld09 output file, instead of a.out.

-v Used to indicate that relevant information about the destinations machines memory configuration reside in a file called "config" within the current directory. The format of such a file is ;

```
[{ram <from hex address> <to hex address>}]
[{rom <from hex address> <to hex address>}]
```

The Rld relocates the object files such that the final executable code image will run in this memory configuration. Zeros fill the non-used address spaces areas in the a.out file to be down loaded.

FILES

/lib/lib?.a09	libraries
a.out	output file

SEE ALSO

a.out(5), ar(1), as09(1), cc09(1).

APPENDIX D - REVIEW OF A CURRENT SYSTEM (AZTEC C65)

The following is an evaluation of the AZTEC "C" system for the Apple IIe produced as an Appendix to "HIGH LEVEL LANGUAGE CODE IMAGES FOR READ ONLY MEMORY" and serves to highlight the problems that can be experienced in real time micro-processor applications which ultimately reside on a ROM.

For the purposes of this work it seemed appropriate to develop a piece of software for ROM using a system currently on the market which claimed to have the capabilities to do so and compare the ease by which this could be achieved using the tools developed during the course of this work.

"AZTEC C65 produces re-entrant code that is ROMable"[1]

The very nature of what is now widely regarded as a primitive and somewhat pedestrian processor in the Rockwell 6502 on which the Apple IIe is based particularly serves to accentuate the problems that exist in the development of High Level Language software for ROM. This is due to the importance of zero-page memory in the Apple IIe and the inherent lack of 16 bit address registers in an eight bit machine.

The application developed can be described as a remote stand alone device based around a 6511 processor (almost identical instruction set to the 6502) , for the collection and

---

1 AZTEC "C" Technical Information Manual - Section 5.3
Page 5-6

transmission of data from both analogue and digital devices which may be connected to the machine.

A primative operating system resides on a 4k EPROM on the mother board of the device. Messages are sent to this remote computer via a standard RS232 interface with instructions concerning the acquisition of Data from any one or more of it's 16 Analogue to Digital 12 bit ports and it's single 16 bit digital input port.

The machine can be told to scan one or more of the A/D ports and when requested to do so, supply the most recent value read from that port.

The bulk of the "operating system" or "monitor" is interrupt driven transmit and receive routines from the RS232. Upon receiving a command string termination character (carriage return) the buffered string is parsed and the machines state is changed to perform some function and return a message or result to the destination machine.

The interrupt routines for this machine amount to approximately one hundred lines of assembler. The main software "work" is performed by the string parsing routine which must interpret the contents of a command string.

The command string parser was seen to be most expediently written in "C" while the interrupt driven serial input and output routines best described in 6502 assembler.

The main problems with the AZTEC "C" system for the Apple IIe was it's prolific use of zero-page memory. Unfortunately the 6511 configuration, unlike that of the Apple IIe, offers zero-page memory from only Hex 40 to Hex FF, the low memory addresses being consumed by I/O ports, control registers and unavailable address space. Another restriction is the fact that the 6511 stack necessarily resides in zero-page and grows down from Hex FF. The resultant memory configuration is one where clearly zero-page address space is at a premium.

The AZTEC "C" compiler makes use of four (4) * 4 byte pseudo registers, a four (4) byte return value from "C" functions, a two byte (2) pseudo stack pointer, a (2) byte pseudo frame pointer all of which must necessarily reside in zero-page memory. Fortunately all of these are simply "equated" in the compilers utility routines (stack save, stack restore etc) and the source code was available on the system.

Thinking that the redefining of these pseudo registers and the re-compilation of these routines was perhaps a simple solution to the problem proved to be somewhat naive.

Upon closer examination of the assembly listings produced as output from the "C" compiler it became apparent that, although the suppliers had kindly allowed the source for the utility routines to be included in the software package they had not supplied the source for the compiler and clearly the pseudo registers where "hard coded" into the system.

Not only was this the case but also some other "pseudo" registers were in use for temporary storage which had not been specified in the system documentation.

The solution then, if one was to persist in using the system, was to compile the "C" language routines and then manually change the pseudo registers hard coded in the system by changing the assembler listings to the zero-page addresses which were acceptable to the destination machines memory configuration.

At this stage it was apparent that the amount of pre-requisite information concerning the internals of the AZTEC "C" system to get this far was far exceeding the effort that had been expended in the development of the monitor software.

Persistence in the evaluation at this point was justified by a curiousity about the amount of effort that would ultimately be needed to produce a ROM executable machine code image using the system and perhaps that the initial effort would pay off given that knowledge of the development system could be used in the long term for future projects. Certainly it would have been more cost effective to have written the entire system in 6502 assembler by this stage.

Perhaps the most irritating aspect of the whole exercise was the AZTEC "C" systems documentation's insistence upon its ability to produce "ROMable" code images by simply offering "loader" options to relocate data and code separately.

"To place a program in ROM, the "-c" option must be

> used [with the loader] to specify the beginning address of the code section of the program. This should correspond to the final address of the ROM in memory [does not make sense]. Likewise, the "-d" option must be used to specify the location of the data section of the program, which in this case will be the address of the available RAM in the system. The "-b" option specifies the base address of the output file and should be set to the lesser of the code and data addresses." 2

This seems like a reasonably sensible approach except for the fact that there is no essential warning concerning the use of zero-page address space or the run time operation of the system which dynamically places the number of arguments passed as well as the argument list address in the first three bytes following the function call !

Furthermore the practicality of the approach is questionable in that;

> "When the modules are linked, both code and data will be placed in the output file with nulls filling the space between. Separating the code and data is up to the user". 3

---

2 Ibid Page 5-6
3 Ibid page 5-6

Imagine the situation if RAM begins a Hex 100 and code begins at Hex F000, as was the case in this prototype system. This means that the loader will generate a executable output file which occupies 63K bytes most of which will be nulls. This is impractical to most Apple IIe users who don't have a hard disk drive and who haven't got several hours to spare while the loading process is carried out (note : try loading a 63K object file into a machine with 64K of address space without destroying the system monitor).

A more acceptable solution would be to discard the data segment entirely, warn the user not to include compile time initialisations in their source code and to simply produce an appropriately relocated machine code image which refers to data variables in RAM and which is not unruly in its size.

Of special interest was the distinction between code and data variables in the assembler listings produced by the compiler. Such a distinction is of primary importance to a system which is to reside on a ROM chip, it being necessary to address the data segment which includes program variables to the RAM space of the machine.

The confusion on the matter of data and code segment separation is even more profound given the following·

> "The startup routine supplied with the libraries is almost certainly unsuitable for a ROM application and must be rewritten. This routine sets up the stack

> pointer and the beginning of available memory for the allocation routines. It must be modified to reflect the configuration of the machine being used. In addition, if there is any initialised data, that data must be either be initialised directly by the startup routine, or a copy of the data can be kept in ROM and copied to RAM as part of the startup routine."4

This seems to be in direct conflict with the preceding. If both code and data segments are produced by the "loader" with nulls filling the gap between the two extents, why is it necessary to initialise data at startup or alternatively, why generate the excessively large code image of both data and code segments in the first place ?

Another anomaly is the use of the word string constants in the discussion;

> "The compiler generates the "cseg" pseudo-op before each section of compiled code to differentiate it from data. String constants are placed in the code segments as well." 5

The concept of a string constant does not exist in "C". A legal expression is;

```
"string"[5] = "t";
```

---

4 Ibid page 5-6
5 Ibid page 5-6

Are "printf" strings, string constants ? Generally the assumption made is a valid one but the flexibility of the language refutes this argument and in practice the implementation of this idea is clumsy, consider the following code generated by the AZTEC "C" compiler along with corresponding "C".

```
*struct key {
*        char *keyword;
*        int  count;
*} keytable[] = {
*        "break",0,
*        "case",0,
*        "char",0,
*        "continue",0,
*        "default",0,
*        "while",0
*};
        cseg
keytable
        asc "break"
        db 0
        fdb 0
        asc "case"
        db 0
        fdb 0
        asc "char"
        db 0
        fdb 0
        asc "continue"
        db 0
        fdb 0
        asc "default"
        db 0
        fdb 0
        asc "while"
        db 0
        fdb 0
```

In this example the defined struct has been included in in the code segment by the compiler although the sub-field of the struct "name" is not a string constant.

It seems doubtful given the above arguments that the implementors of the AZTEC "C" system went to the trouble to produce

ROM executable code for a "real" system which works. If they have then they have done so either using a very trivial example on a machine whose memory configuration (especially zero-page memory) is not unlike that of the standard Apple IIe running DOS. Alternatively they have done so with access to the full source code of the compiler and the loader which was modified to suit the customised needs of the system developed.

One likely reason for the inclusion of a discussion of "ROMable code" is increase the credibility of the system with technical users who have no intention of running stand alone systems from ROM but are impressed by software which claims it can do so.

In any case the suitability of the system for producing stand alone customised micro-computer systems which reside on ROM is, at best doubtful, and at worst impossible.

```
Flush_Rec(record)
RECTYPE *record ;
{
        if (record.rsize != 0)
                mmput(coutb,(char *)record,
                   (record->rsize
                    + sizeof record->location
                    + sizeof record->rsize)
                    );
}
Compress_Seg(buf,c,origin)
short *buf;
char c;
unsigned short origin;
{
        register bool rec = NO;
        register int loc,i,j,max=128,nullcount=0;
        char buffer[128];

        flush(buf);
        close(buf[0]);
        tfname[6] = c;
```

```
if (!mgetfile(tfname))
        error(1, "cannot open");
else {
        loc = origin;
        dseek(&text,0,0,Dsize);
        while (text.size > 0) {
            if (text.size < max)
                max = text.size;
            mget((char *)&buffer,max);
            for (i=0; i< max; i++) {
                if (buffer[i] == NULL) {
                        loc++;
                        nullcount++;
                        continue;
                }
                if (nullcount >= 4 ) {
                        if (rec) {
                            Flush_Rec(&record);
                                rec = NO;
                        }
                }
                if (rec) {
                        if (record.rsize == MAX_DATA) {
                                Flush_Rec(&record);
                                rec = NO;
                                i--;
                        }
                        else {
                            if ((nullcount>0)&&
                                (nullcount<4)&&
                                (record.rsize>0))
                                    for(j=i-nullcount;j < i; j++)
                                        record.data[record.rsize++]=NULL;
                            record.data[record.rsize++] = buffer[i];
                            loc++;
                        }
                }
                else {
                        record.location = loc;
                        record.rsize = 0;
                        i--;
                        rec = YES;
                }
                nullcount = 0;
        }
    }
    if (rec)
        Flush_Rec(&record);
    record.rsize = 0;
    Flush_Rec(&record);
}
```

APPENDIX F - Compressed Data Segment Estimation Code

```
/*
 * make some estimate of the amount of bytes that will be
 * able to be compressed in this code images data segment
 * by examination of the data and relocation bits in each .o file
 */

extern RECTYPE record;
extern CHAR_SIZE;
extern STREAM reloc;

/*
 *       returns the estimated size of the compacted data segment
 *       for a .o mod
 *
 *
 */

Estimate( bno, off)
{
     register short t,r ;
     static loc_size = sizeof record.location,
     rsize_size = sizeof record.rsize,
     T_R_SIZE = sizeof t;
     register accum=0 ,
     MAX = MAX_DATA * (CHAR_SIZE / T_R_SIZE);
     register bool swt;
     static bool lastnull=TRUE;
```

```
Knap()
{
  if (solvable()) {
     allocfit(chunk);
     /* simply allocate the chunks to spaces */
     return;
  }
  else
     error(0,"no solution exists suggest slitting source files");
}
solvable()
{
  if (!Chk_Space_sizes()) {
     error(0,"Spaces overlap");
     return(0);
  }
  switch(check1()) {
  case REDUCE_CHUNK:
     error(0,"reduce object size input file");
     return(0);
  case SMALL:
     error(0,"memory too small for code");
     return(0);
  case OK:
     if (tryfit(chunk))        /* in the order given */
        return(GOOD);
     sorts(space,no_spaces); /* sort spaces in decsending size */
     sort(chunk,no_chunks);  /* sort chunks in decsending size */
     if (tryfit(chunk))
        return(GOOD);
     return(permut(0));        /* otherwise try all permuations */
  }
}
```

```
permut(k)
int k;
{
  int i,j,temp,old_fail_cnt,wait_cnt;

  for (j=k; j< no_chunks ; j++) {
     swap(&chunk[k].ch_size,&chunk[j].ch_size);
     if (k < no_chunks-1)
        if (permut(k+1)) return(GOOD);
     else {
        if (tryfit(chunk)) {
           good_cnt++;
           wait_cnt = fail_cnt-old_fail_cnt;
           tot_wait += wait_cnt;
           old_fail_cnt = fail_cnt;
           if (wait_cnt > max_wait)
              max_wait = wait_cnt;
           return(GOOD);
        }
        else {
           fail_cnt++;
           continue;
        }
     }
     swap(&chunk[k].ch_size,&chunk[j].ch_size);
  }
  return(BAD);
}


tryfit(x)
struct CHUNK x[];
{
  int i,temp=0;
  int ind= 0;
  for (i=0;i<no_chunks;i++) {
     if ((temp=temp-x[i].ch_size) >= 0) continue;
     if (ind == no_spaces)
        return(0);
     if ((temp=space[ind++].sp_size-x[i].ch_size) >= 0) continue;
     else {
        i--;
     }
  }
  if (ind > no_spaces)
     return(BAD);
  return(GOOD);
}
```

```
allocfit(x)
struct CHUNK x[];
{
  int i,temp=0;
  int ind= 0;
  int addr;

  for (i=0;i<no_chunks;i++) {
     if ((temp=temp-x[i].ch_size) >= 0) {
        x[i].ch_space_no = ind;
        x[i].ch_start = addr;
        addr += x[i].ch_size;
        continue;
     }
     if ((temp=space[ind].sp_size-x[i].ch_size) >= 0) {
        addr = space[ind++].start;
        x[i].ch_space_no = ind;
        x[i].ch_start = addr;
        addr += x[i].ch_size;
        continue;
     }
     else
        i--;
  }
}


/*      make sure that no one object unit exceeds the largest space size
        check that there's enough rrrom for all object units in the space
*/
check1()
{
  int max_chunk=0,max_space=0,space_size=0,chunk_size=0,i;

  for (i=0; i<no_spaces; i++) {
     space_size += space[i].sp_size;
     if (space[i].sp_size > max_space)
        max_space=space[i].sp_size;
  }
  for (i=0; i<no_chunks; i++) {
     chunk_size += chunk[i].ch_size;
     if (chunk[i].ch_size > max_chunk)
        max_chunk = chunk[i].ch_size;
  }
  if (chunk_size > space_size)
     return(SMALL);

  if (max_chunk > max_space)
     return(REDUCE_CHUNK);

  return(OK);
}
```

```
/*
   check that spaces don't overlap
*/
Chk_Space_size()
{
  int i,j,start,end;
  for (j=0; j< no_spaces; j++) {
     start = space[j].start;
     end = space[j].end;
     for (i=j+1; i<no_spaces ; i++)
        if (space[i].start < end)
           return(BAD);
  }
  return(GOOD);
}


/* sorts chunks by size */
sort(v,n)
struct CHUNK v[];
{
  int temp,gap,i,j;

  for (gap= n/2; gap>0; gap/=2)
     for (i=gap;i<n;i++)
        for(j=i-gap;j >=0; j-=gap) {
           if (v[j].ch_size > v[j+gap].ch_size)
                break;
        swap_ch(v[i],v[i+gap]);
        }
}
/* sorts spaces by size */
sorts(v,n)
struct SPACE v[];
{
  int temp,gap,i,j;

  for (gap= n/2; gap>0; gap/=2)
     for (i=gap;i<n;i++)
        for(j=i-gap;j >=0; j-=gap) {
           if (v[j].sp_size > v[j+gap].sp_size)
              break;
        swap_sp(v[i],v[i+gap]);
        }
}
```